

Classes

- **What Are Classes?**

Classes extend the built-in capabilities of C++ to assist you in representing and solving complex, real-world problems.

You make a new type by declaring a class. A class is just a collection of variables combined with a set of related functions.

A class can consist of any combination of the variable types and also other class types. The variables in the class are referred to as the *member variables* or *data members*. The functions in the class typically manipulate the member variables. They are referred to as *member functions* or *methods* of the class.

A class enables you to encapsulate these variables and functions into one collection, which is called an *object*.

- **Declaring a Class**

To declare a class, use the **class** keyword followed by an opening brace, and then list the data members and methods of that class. End the declaration with a closing brace and a semicolon.

The following code is the general syntax for declaring a class.

```
class ClassName
{
    memberList
};
```

Here *ClassName* is the name of the class, and *memberList* is the list of class members.

memberList consists of a list of member declarations. These can be data member or method declarations.

Data member declarations are normal variable declarations. For example, `int x;` is a data member declaration when inside a class. However, you cannot initialize data members where you declare them. They must be initialized either in a method or outside the class. For example, `int x = 50;` as a data member declaration causes an error.

Method declarations are function declarations placed inside a class (recall that a function declaration can also include the implementation, or you can implement it separately). All methods can be accessed only through an object of the class.

Example:

```
class Part
{
    int modelnumber;
    double cost;
    void SetPart(int mn, double c);
    void ShowPart();
};
```

- **Defining an Object**

An *object* is an individual instance of a class. You define an object of your new type just as you define an integer variable (or any other variable):

```
Part wheel;
```

This code defines `wheel`, which is an object whose class (or type) is `Part`.

- **Calling Data Member and Member Functions**

Once you define an actual `Part` object -for example, `wheel`- you use the dot operator (.) to access the members of that object.

Therefore, to assign 50 to `wheel`'s `modelnumber` member variable, you would write

```
wheel.modelnumber = 50;
```

In the same way, to call the `ShowPart()` function, you would write

```
wheel.ShowPart();
```

When you use a class method, you call the method. In this example, you are calling `ShowPart()` on `wheel`.

- **Assign Values to Objects, Not to Classes**

In C++ you don't assign values to types; you assign values to variables. For example, you would never write

```
int = 50; //wrong
```

The compiler would flag this as an error, because you can't assign 50 to an integer. Rather, you must define an integer variable and assign 50 to that variable. For example,

```
int x; //define x to be an int
```

```
x = 50; //set x's value to 50
```

This is a shorthand way of saying, "Assign 50 to the variable **x**, which is of type **int**".

In the same way, you wouldn't write

```
Part.modelnumber = 50; //wrong ???
```

The compiler would flag this as an error, because you can't assign 50 to the `modelnumber` of a `Part`. Rather, you must define a `Part` object and assign 50 to the `modelnumber` of that object.

For example:

```
Part wheel; //just like int x;  
wheel.modelnumber = 50; //just like x = 50;
```


- **Using Access Specifiers**

C++ allows you to control where the data members of your class can be accessed. This control is a powerful tool because it allows you to protect data members from accidental. An *access specifier* is a word that controls where the data members in a class can be accessed. The syntax for an access specifier is as follows:

```
class ClassName
{
    classMembers;
    accessSpecifier:
    classMembers;
};
```

An access specifier affects all members of the class (including methods) that come after it until another access specifier is encountered or until you reach the end of the class.

A class has two kinds of access specifiers: **public** and **private**.

- *public members* can be accessed anywhere that an object of the class can be accessed and from within the class (that is, in the class's methods).
- *private members* can be accessed only from within the class itself. An object of the class cannot access the private members, except through public methods. If no access specifier is provided in the class, all members default to private.

Here is an example of how you might use these specifiers:

```
class ClassName
{
    int x;
public:
    int y;
    int z;
private:
    int a;
};
```

In this example, `x` and `a` are private members, and `y` and `z` are public. You may have as many access specifiers as you want in your classes.

- **Class Members Are Private by Default**

In the class `Part` declaration example `modelnumber`, `cost`, `SetPart()`, and `ShowPart()` are all private, because all members of a class are private by default. This means that unless you specify otherwise, they are private. Therefore, if you write

```
Part wheel;  
wheel.modelnumber = 50; //error
```

the compiler flags this as an error, because `modelnumber` is private, and you can't access private data.

The way to use `wheel` so that you can access the data members is to declare it **public**.

```
class Part
{
public:
    int modelnumber;
    double cost;
    void SetPart(int mn, double c);
    void ShowPart();
};
```

Now modelnumber, cost, SetPart(), **and** ShowPart() **are** all **public**. And wheel.modelnumber = 50; **compiles without** problems.

- **Private and Public Data**

Usually the data within a class is **private** and the functions are **public**. This is a result of how classes are used. The data is hidden so it will be safe from accidental manipulation, while the functions that operate on the data are public so they can be accessed from outside the class. However, there is no rule that data must be private and functions public; in some circumstances you may find you'll need to use private functions and public data.

```
class Part
{
private:
    int modelnumber;
    double cost;
public:
    void SetPart(int mn, double c);
};
```

```
#include <iostream.h>
class Part
{
private:
    int modelnumber;
    double cost;
public:
    void SetPart(int mn, double c) //set data
    {
        modelnumber = mn;
        cost = c;
    }
    void ShowPart() //display data
    {
        cout << "Model " << modelnumber;
        cout << ", costs $" << cost << endl;
    }
};
```

```
int main()  
{  
    Part part1, part2, part3;  
    part1.SetPart(6245, 220);  
    part2.SetPart(6246, 185);  
    part3.SetPart(6247, 150);  
    part1.ShowPart();  
    part2.ShowPart();  
    part3.ShowPart();  
    return 0;  
}
```

Output:

```
Model 6245, costs $220  
Model 6246, costs $185  
Model 6247, costs $150
```


• **Memory Allocation**

Declaring this class doesn't allocate memory for a **Part** . It just tells the compiler what a **Part** is, what data it contains (**modelnumber** and **cost**), and what it can do (**SetPart()** and **ShowPart()**). It also tells the compiler how big a **Part** is (that is, how much room the compiler must set aside for each **Part** that you create). In short, specifying a basic type does two things:

- It determines how much memory is needed for a data object.
- It determines what operations, or methods, can be performed using the data object.

Each new object you create contains storage for its own internal variables, the class members. But all objects of the same class share the same set of class methods, with just one copy of each method.

- **Creating Methods**

Each function that you declare for your class must have a definition. The definition is also called the function implementation. Like other functions, the definition of a class method has a function header and a function body.

You can declare a method two ways. The most common way is to declare a method inside the class declaration and then implement it outside. The second way is to declare and implement the method at the same time inside the class declaration.

You should declare most of your methods the first way. However, there is special syntax for the method definition. A member function definition begins with the return type, followed by the name of the class, two colons (: :), the name of the function, and its parameters.

Here is the general syntax for a method implementation that occurs outside a class:

```
return_type ClassName::methodName(parameterList)  
{  
    methodImplementation;  
}
```

Here *ClassName* is the name of the class, and *methodImplementation* is the code that goes inside the method. As you can see, this syntax is very similar to the function definition syntax. The double colon (`::`) is called the *scope resolution operator*. Here you are telling the computer to use the class's scope by putting the class's name in front.

- **Constructor and Destructor**

Two special kinds of methods, the constructor and the destructor, can be in a class. Both are optional, but they provide special functionality that other methods cannot provide.

A constructor is executed every time a new instance of the class is created -that is, every time you declare a new object-. The constructor is normally used to set initial values for the data members. A constructor always has the same name as the class and cannot have a return value (not even **void**).

A destructor is the opposite of a constructor and is executed when the object is destroyed. The destructor is always named the same name as the class, but with a tilde (~) at the beginning. The destructor cannot have arguments or a return value. A destructor is often used to perform any necessary cleanup tasks.

Both constructors and destructors are like methods; they can be declared and implemented at the same time or declared and implemented separately. Here is the syntax for declaring and implementing at the same time:

```
class ClassName
{
    //constructor
    ClassName( [argumentList] )
    {
        implementation;
    }
    //destructor
    ~ClassName( )
    {
        implementation;
    }
    //other members
};
```

Here is the syntax for declaring and then implementing:

```
class ClassName
{
    ClassName( [argumentList] );
    ~ClassName( );
    //other Members
};
```

```
ClassName::ClassName( [argumentList] )
{
    implementation;
}
```

```
ClassName::~ClassName( )
{
    implementation;
}
```

Notice that the constructor can have arguments. If you create a constructor with arguments, the user of your class must supply values for these arguments when creating an object.

The destructor, on the other hand, cannot have arguments. It is called automatically, so there isn't necessarily a chance for the user to provide arguments.

Because a constructor can have arguments, it might become necessary to overload the constructor. This is legal in C++ and is quite common in large classes. Overloading the constructor in this way gives your class versatility and provides users of the class with many options.

The destructor cannot be overloaded. Having no return type or arguments, there is nothing with which the destructor can be overloaded.

```
// Example 1: constructor without parameters
#include <iostream.h>
class Part
{
private:
    int modelnumber, quantity;
    double cost;
public:
    Part() //constructor - no parameters
    {
        quantity = 100;
    }
    ~Part(){} //destructor
    void SetPart(int mn, double c)
    {
        modelnumber = mn;
        cost = c;
    }
}
```



```
void ShowPart()  
{  
    cout << "Model " << modelnumber;  
    cout << ", quantity " << quantity;  
    cout << ", costs $" << cost << endl;  
}  
};  
int main()  
{  
    Part part1, part2;  
    part1.SetPart(6245, 220);  
    part2.SetPart(6246, 185);  
    part1.ShowPart();  
    part2.ShowPart();  
    return 0;  
}
```

Output:

```
Model 6245, quantity 100, costs $220  
Model 6246, quantity 100, costs $185
```

```
// Example 2: constructor with parameters
#include <iostream.h>
class Part
{
private:
    int modelnumber, quantity;
    double cost;
public:
    Part(int q) //constructor with parameter
    {
        quantity = q;
    }
    ~Part(){} //destructor
    void SetPart(int mn, double c)
    {
        modelnumber = mn;
        cost = c;
    }
}
```

```
void ShowPart()  
{  
    cout << "Model " << modelnumber;  
    cout << ", quantity " << quantity;  
    cout << ", costs $" << cost << endl;  
}  
};  
int main()  
{  
    Part part1(320), part2(240);  
    part1.SetPart(6245, 220);  
    part2.SetPart(6246, 185);  
    part1.ShowPart();  
    part2.ShowPart();  
    return 0;  
}
```

Output:

```
Model 6245, quantity 320, costs $220  
Model 6246, quantity 240, costs $185
```

```
// Example 3: constructor with overloading
#include <iostream.h>
class Part
{
private:
    int modelnumber, quantity;
    double cost;
public:
    Part() //constructor - no parameters
    {
        quantity = 100;
    }
    Part(int q) //constructor with parameter
    {
        quantity = q;
    }
    ~Part(){} //destructor
    void SetPart(int mn, double c)
    {
        modelnumber = mn;
        cost = c;
    }
}
```

```

void ShowPart()
{
    cout << "Model " << modelnumber;
    cout << ", quantity " << quantity;
    cout << ", costs $" << cost << endl;
}
};
int main()
{
    Part part1, part2(200);
    part1.SetPart(6245, 220);
    part2.SetPart(6246, 185);
    part1.ShowPart();
    part2.ShowPart();
    return 0;
}

```

Output:

```

Model 6245, quantity 100, costs $220
Model 6246, quantity 200, costs $185

```

- **Separating Classes into Files**

Classes often get pretty big, and having all your code in one file can quickly become unmanageable. Also, if you want to reuse your classes in other programs, you have to copy and paste them into the new program.

Fortunately, there is a convention for separating classes into files. Normally, the class declaration is placed in one file (header file), and the implementation of all the methods is put in another file. The class declaration file is normally called `ClassName.h`, where `ClassName` is the name of the class. The implementation file is normally called `ClassName.cpp`. Then you include the header file in your program with an `#include` directive.

The syntax for the `#include` directive is as follows:

```
#include "filename"
```

However, instead of including two files (ClassName.h and ClassName.cpp), you have to include only ClassName.h. The compiler will include the .cpp file automatically (provided that the two files are in the same directory).

Here is how the Part class looks separated into different files:

```
//Part.h
class Part
{
private:
    int modelnumber, quantity;
    double cost;
public:
    Part(int q);
    ~Part();
    void SetPart(int mn, double c);
    void ShowPart();
};
```

```

//Part.cpp
#include <iostream.h>
#include "Part.h"

Part::Part(int q)
{
    quantity = q;
}
Part::~Part()
{
}
void Part::SetPart(int mn, double c)
{
    modelnumber = mn;
    cost = c;
}
void Part::ShowPart()
{
    cout << "Model " << modelnumber;
    cout << ", quantity " << quantity;
    cout << ", costs $" << cost << endl;
}

```



```
// Main.cpp
#include "Part.h"

int main()
{
    Part part1(150), part2(200);
    part1.SetPart(6245, 220);
    part2.SetPart(6246, 185);
    part1.ShowPart();
    part2.ShowPart();
    return 0;
}
```

Output:

```
Model 6245, quantity 150, costs $220
Model 6246, quantity 200, costs $185
```

- **Pointers To Objects**

An object of a class has a memory address—just like any other object. You can assign this address to a suitable pointer.

Example:

```
Account savings("Mac, Rita", 654321, 123.5);  
Account *ptrAccount = &savings;
```

This defines the object `savings` and a pointer variable called `ptrAccount`. The pointer `ptrAccount` is initialized so that it points to the object `savings`. This makes `*ptrAccount` the object `savings` itself. You can then use the statement

```
( *ptrAccount ).display();
```

to call the method `display()` for the object `savings`. Parentheses must be used in this case, as the operator `.` has higher precedence than the `*` operator.

- **Arrow Operator**

You can use the class member access operator `->` (in short: arrow operator) instead of a combination of `*` and `.`

Syntax:

`objectPointer->member`

This expression is equivalent to

`(*objectPointer).member`

The operator `->` is made up of a minus sign and the greater than sign.

Example:

`ptrAccount->display();`

This statement calls the method `display()` for the object referenced by `ptrAccount`, that is, for the object `savings`. The statement is equivalent to the statement in the previous example.

The difference between the class member access operators (.) and (->) is that the left operand of the dot operator must be an object, whereas the left operand of the arrow operator must be a pointer to an object.